Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій та моделювання

Кваліфікаційна робота
за освітнім ступенем «бакалавр»
на тему:

ІГРОВИЙ ВЕБДОДАТОК, РОЗГОРНУТИЙ НА ХМАРНІЙ ПЛАТФОРМІ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ КОНТЕЙНЕРИЗАЦІЇ

Виконав:
здобувач IV курсу
групи ІПЗ-41
спеціальності 121 «Інженерія
програмного забезпечення»
Бенца Святослав В’ячеславович

Науковий керівник:
к. т. н., доцент Шевцова Н. В.

Рівне – 2025

ЗМІСТ

ВСТУП.....3
РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ КОНТЕЙНЕРНИХ ТЕХНОЛОГІЙ.....5
1.1. Принципи роботи контейнерів.....5
1.2. Порівняльний аналіз віртуалізації та контейнеризації.....7
1.3. Основні переваги та недоліки контейнеризації.....9
РОЗДІЛ 2. ПРОЕКТУВАННЯ ІГРОВОГО ВЕБДОДАТКУ З УРАХУВАННЯМ КОНТЕЙНЕРИЗАЦІЇ.....13
2.1 Аналіз вимог та структура застосунку на основі контейнерної архітектури.....13
2.2. Вибір технологій для Frontend частини вебзастосунку.....17
2.3. Вибір технологій для Backend частини вебзастосунку.....21
РОЗДІЛ 3. ТЕХНІЧНА РЕАЛІЗАЦІЯ ІГРОВОГО ВЕБДОДАТКУ.....26
3.1. Налаштування інфраструктури розробки з використанням контейнерних технологій.....26
3.2. Створення та налаштування бази даних PostgreSQL у контейнері.....29
3.3. Використання WebSocket для реалізації ігрової логіки з розподілом функціоналу по сервісах.....32
3.4. Структура клієнтської частини вебдодатку.....37
РОЗДІЛ 4. РОЗГОРТАННЯ ТА ТЕСТУВАННЯ КЛІЄНТ-СЕРВЕРНОЇ ГРИ.....40
4.1. Вибір хмарної інфраструктури для ігрового вебзастосунку.....40
4.2. Тестування гри в умовах продакшн-середовища на хмарній платформі.....42
ВИСНОВКИ.....45
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....47

ВСТУП

Актуальність дослідження. Сучасна розробка вебдодатків все частіше орієнтується на використання хмарних технологій та контейнеризації, що забезпечують гнучкість, масштабованість і ефективне використання ресурсів. Особливо актуальним це стає для ігрових вебдодатків, які функціонують у режимі реального часу та мають підвищені вимоги до продуктивності та надійності. Технологія контейнеризації, зокрема Docker, у поєднанні з хмарними сервісами дозволяє спростити процес розгортання, оптимізувати витрати на інфраструктуру та забезпечити безперебійну роботу додатків навіть при змінних навантаженнях.

Мета дослідження. Розробити ігровий вебдодаток, який функціонує в режимі реального часу, та розгорнути його на хмарній платформі з використанням технології контейнеризації.

Завдання дослідження:

– проаналізувати особливості використання контейнеризації в хмарних середовищах;

– дослідити можливості хмарних платформ для розгортання вебдодатків;

– розробити архітектуру ігрового вебдодатка;

– реалізувати клієнтську та серверну частини додатка;

– забезпечити взаємодію в реальному часі за допомогою вебсокетів;

– виконати контейнеризацію та розгортання додатка на хмарній платформі;

– оцінити ефективність обраного підходу.

Об’єкт дослідження. Хмарні сервіси для розгортання вебдодатків з використанням контейнерів.

Предмет дослідження. Застосування технологій контейнеризації та хмарних платформ для створення ігрових вебдодатків.

Методи дослідження. Аналіз літературних джерел, моделювання, проектування програмної архітектури, практична реалізація, тестування продуктивності, порівняльний аналіз.

Практичне значення дослідження. Результати дослідження можуть бути використані для розробки масштабованих та продуктивних вебдодатків, які функціонують у реальному часі та легко розгортаються на різних хмарних платформах.

Апробація та впровадження результатів дослідження. Результати дослідження доповідалися на звітній конференції здобувачів освіти та викладачів РДГУ 15 травня 2025 року. Було продемонстровано вебдодаток, зокрема його користувацький інтерфейс, основні ігрові функції, роботу backend-сервера, а також взаємодію з базою даних. Було показано, як відбувається обробка запитів, підключення через WebSocket, збереження даних у PostgreSQL, а також масштабування проєкту за допомогою Docker у хмарному середовищі.

Структура роботи. Робота складається зі вступу, чотирьох розділів, висновків та списку використаних джерел. У першому розділі проаналізовано теоретичні основи контейнеризації та хмарних сервісів. У другому – описано проєктування ігрового вебдодатка. Третій розділ присвячено реалізації клієнтської та серверної частин. У четвертому – описано процес розгортання додатка у хмарному середовищі та оцінено його ефективність.

РОЗДІЛ 1
ОГЛЯД КОНТЕЙНЕРНИХ ТЕХНОЛОГІЙ

1.1. Принципи роботи контейнерів

У сучасних підходах до розробки та розгортання програмного забезпечення дедалі ширше застосовується контейнеризація. Цей підхід передбачає об’єднання програмного застосунку з усіма необхідними компонентами (програмним кодом, бібліотеками, конфігураційними файлами) в єдиний ізольований контейнер [5]. Така ізоляція забезпечує стабільність і передбачуваність роботи програмного забезпечення незалежно від середовища його запуску. Використання контейнерів дає змогу уникнути повторного ручного налаштування середовища на кожному сервері, оскільки контейнер функціонує як самодостатній і портативний об’єкт для розгортання.

Контейнери слід чітко відрізняти від віртуальних машин (ВМ), оскільки між цими підходами існують принципові архітектурні відмінності. Віртуальні машини імітують повноцінне апаратне середовище, включаючи окреме ядро операційної системи, що призводить до значного споживання ресурсів і тривалого часу запуску. Натомість контейнери функціонують на рівні операційної системи хоста, використовуючи її ядро спільно з іншими контейнерами. Такий підхід забезпечує більш ефективне використання ресурсів і суттєве скорочення часу ініціалізації.

Ізоляція контейнерів досягається завдяки системним механізмам ядра Linux, зокрема просторам імен (namespaces) та контрольним групам (cgroups). Ці механізми забезпечують чітке розмежування ресурсів, таких як файлові системи, мережеві інтерфейси, оперативна пам’ять і процесорний час, що дозволяє кільком контейнерам функціонувати незалежно один від одного в межах одного хосту.

Ключовими перевагами контейнеризації є її гнучкість і легкість інтеграції в існуючі IT-інфраструктури. Розгортання контейнера починається з образу (image) – спеціального шаблону, який містить усі необхідні компоненти для виконання програмного забезпечення (див. рисунок 1.1). У результаті, при запуску контейнера програма відразу працює у попередньо налаштованому середовищі, що мінімізує потребу в додатковій конфігурації, прискорює процес запуску і загалом спрощує архітектуру системи.

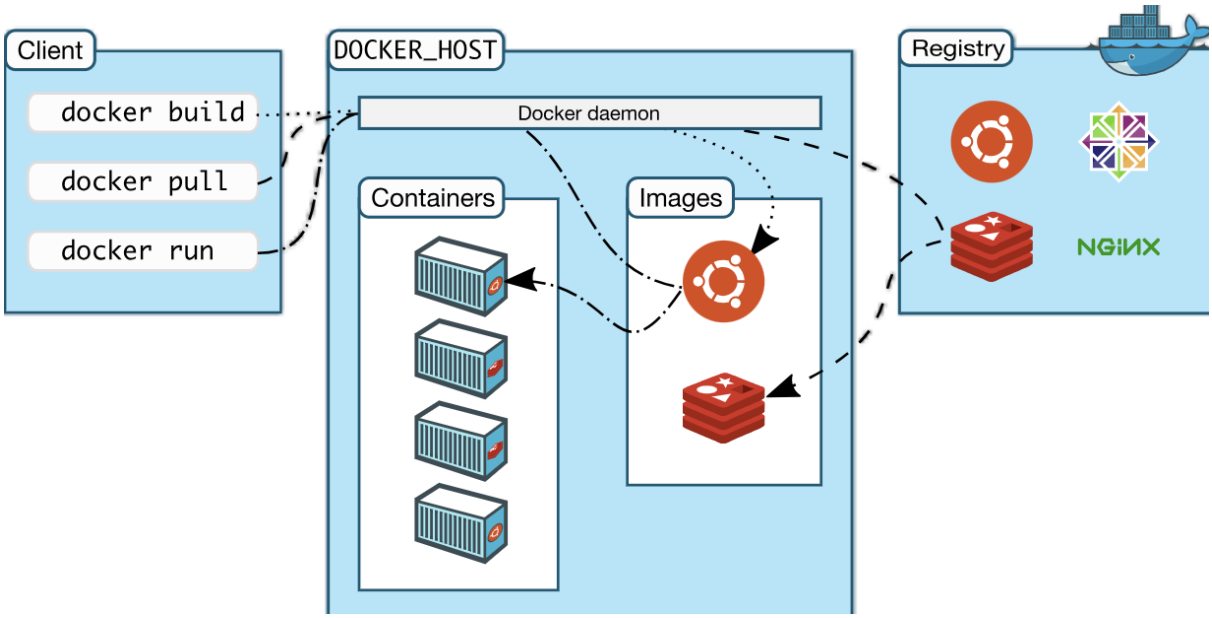


Рисунок 1.1 Схема роботи докера і докер контейнерів

Таким чином, контейнеризація – це сучасний метод організації програмного забезпечення, який забезпечує ізоляцію компонентів у самодостатні середовища, що включають усі залежності: програмний код, бібліотеки, конфігураційні файли тощо. Однією з її головних переваг є висока портативність: застосунки можуть працювати стабільно в будь-якому середовищі, незалежно від його специфіки, без впливу на базову операційну систему хоста [1]. У контексті життєвого циклу розробки програмного забезпечення (SDLC) контейнеризація стала важливим інструментом, що

дозволяє зменшити кількість помилок і конфліктів між етапами розробки, тестування та розгортання.

Контейнеризація з Docker дозволяє зменшити час на розгортання додатків, підвищити їхню портативність та забезпечити стабільність роботи в різних середовищах. Це робить Docker важливим інструментом у сучасних процесах розробки програмного забезпечення та DevOps практиках [25].

1.2. Порівняльний аналіз віртуалізації та контейнеризації.

Віртуалізація – це технологія, яка забезпечує можливість одночасного запуску кількох ізольованих операційних систем на одному фізичному сервері. Завдяки віртуальним машинам (ВМ), кожна з яких функціонує як самостійний комп'ютер з власною гостьовою операційною системою, процесами, пам'яттю та файловою системою, досягається високий рівень гнучкості у використанні обчислювальних ресурсів [4]. Управління цими віртуалізованими середовищами здійснюється за допомогою гіпервізора – спеціалізованого програмного забезпечення, що виконує розподіл ресурсів між віртуальними машинами, забезпечуючи їхню ізоляцію та стабільність.

Попри переваги, віртуалізація має певні обмеження, зокрема суттєве споживання ресурсів та відносно тривалий час запуску, зумовлений необхідністю завантаження повноцінної операційної системи для кожної ВМ. Крім того, управління великою кількістю віртуальних машин у масштабованому середовищі може бути складним завданням.

Контейнеризація, на відміну від віртуалізації, використовує принципи ізоляції процесів на рівні операційної системи хоста без потреби у запуску повноцінної гостьової ОС для кожного середовища. Контейнери спільно використовують ядро хостової системи, залишаючись при цьому ізольованими один від одного. Вони включають лише необхідні компоненти для виконання

конкретного застосунку, що робить їх надзвичайно легкими, швидкими в запуску та зручними для перенесення.

Одним із найпоширеніших інструментів для роботи з контейнерами є Docker, який забезпечує створення, управління та розгортання контейнерів на основі попередньо сформованих образів (images). Образ – це шаблон, що містить усі необхідні залежності, конфігурації та виконувані файли. Завдяки цьому контейнер може бути запущений практично миттєво, незалежно від середовища, у якому він розгортається [15]. Така архітектура ідеально підходить для реалізації сучасних DevOps-підходів і мікросервісної архітектури.

Основу контейнеризації в Linux-середовищі становлять механізми namespaces та control groups (cgroups). Перші забезпечують логічну ізоляцію – кожен контейнер має окремий простір процесів, мережі, файлової системи та користувачів [6]. У свою чергу, cgroups дозволяють точно обмежувати та контролювати використання ресурсів, таких як пам’ять, процесорний час або введення/виведення, що забезпечує ефективне управління навіть у високонавантажених системах.

Контейнери створюються з багатошарових образів, що оптимізує зберігання даних і спрощує оновлення – кожен шар відповідає за окрему частину конфігурації або коду. При запуску контейнера формується окремий ізольований екземпляр, який, незважаючи на автономність, може взаємодіяти з іншими контейнерами через віртуальні мережі. Мережеві механізми дозволяють створювати логічні канали обміну даними, імітуючи роботу фізичних мереж. Для забезпечення збереження важливих даних, що мають зберігатися незалежно від життєвого циклу контейнера, використовуються зовнішні томи або механізми постійного зберігання.

У масштабних розгортаннях контейнерна архітектура інтегрується з системами оркестрації (наприклад, Kubernetes), які дозволяють автоматизувати управління контейнерами: їх створення, масштабування, оновлення та

моніторинг [3]. Це забезпечує створення гнучких, стійких до відмов інфраструктур, які відповідають вимогам сучасного програмного забезпечення щодо надійності, масштабованості та ефективності використання ресурсів.

1.3. Основні переваги та недоліки контейнеризації

Контейнеризація стала однією з ключових технологій у розвитку сучасних програмних систем, забезпечуючи новий рівень ефективності, гнучкості та масштабованості при розробці, тестуванні та розгортанні програмного забезпечення. Однією з її основних переваг є портативність: кожен контейнер включає всі необхідні залежності та середовище виконання для роботи застосунку, що дозволяє безперешкодно переносити його між різними обчислювальними середовищами – від локальних машин розробників до масштабованих хмарних платформ. Це значно знижує ризики, пов’язані з розбіжностями у середовищах розробки та продуктивного розгортання, і спрощує процес доставки програмного забезпечення.

Іншою вагомою перевагою є ізоляція контейнерів. Кожен контейнер функціонує у власному, ізольованому середовищі, що унеможливлює конфлікти між бібліотеками, залежностями або конфігураціями окремих застосунків [4]. Така ізоляція підвищує стабільність і передбачуваність роботи системи загалом. Крім того, на відміну від традиційних віртуальних машин, контейнери запускаються значно швидше та мають менші вимоги до ресурсів, що дозволяє ефективно використовувати обчислювальні потужності та динамічно масштабувати сервіси.

Контейнерні технології тісно інтегруються з сучасними підходами в розробці програмного забезпечення, зокрема DevOps та CI/CD (Continuous Integration / Continuous Deployment). Контейнери забезпечують стандартизоване середовище виконання, що суттєво спрощує автоматизацію процесів тестування, збірки, доставки й оновлення застосунків [25]. Вони дозволяють

зменшити час, необхідний для розгортання нових версій програм, і знижують ймовірність помилок, спричинених відмінностями середовищ. У поєднанні з системами оркестрації, такими як Kubernetes, контейнерні технології забезпечують можливість побудови самовідновлюваних, автоматизованих і масштабованих програмних архітектур. Такі системи здатні динамічно балансувати навантаження, автоматично відновлювати збої та забезпечувати безперервну доступність сервісів.

Попри очевидні переваги, контейнеризація має і певні обмеження. Основним з них є потенційні ризики безпеки, зумовлені спільним використанням ядра операційної системи. У разі неправильних налаштувань або наявності вразливостей в одному з контейнерів, існує ризик впливу на інші контейнери або навіть на хостову систему. І хоча сучасні механізми ізоляції (наприклад, seccomp, AppArmor, SELinux) частково знижують ці ризики, вони не забезпечують повної ізоляції, як у випадку з віртуальними машинами. Крім того, ефективне впровадження контейнерних технологій потребує глибокого розуміння архітектурних принципів та володіння відповідними інструментами, що може ускладнити вхід до технології для початківців. У великих проєктах виникає необхідність у додаткових компонентах, таких як системи оркестрації, засоби моніторингу, зберігання станів тощо, що значно ускладнює інфраструктуру.

Центральне місце в технології контейнеризації займає інструмент Docker, який з часу свого появлення став не лише найпопулярнішим середовищем для створення та запуску контейнерів, але й основою для розгортання цілісної технологічної платформи, яка охоплює весь життєвий цикл програмного забезпечення – від розробки до масштабованого розгортання [1].

Одним із ключових компонентів технологічного середовища є Docker Hub – централізоване репозиторне сховище контейнерних образів, що забезпечує швидкий доступ до офіційних образів популярних інструментів і платформ (наприклад, NGINX, PostgreSQL, Node.js тощо). Docker Hub дозволяє

не лише завантажувати готові образи, а й розміщувати власні – як у відкритому доступі, так і в приватних репозиторіях. У поєднанні з Docker CLI, цей інструмент забезпечує просте управління життєвим циклом образів на локальних і віддалених системах.

Для опису та запуску багатоконтейнерних застосунків використовується Docker Compose – інструмент, який дозволяє створювати декларативні конфігурації у форматі YAML. У таких конфігураційних файлах описуються всі необхідні сервіси, залежності між ними, змінні середовища, томи та мережеві зв'язки [16]. Це особливо актуально для локальної розробки та тестування мікросервісних застосунків, які складаються з великої кількості окремих компонентів.

У промислових сценаріях розгортання критично важливу роль відіграють системи оркестрації, головним представником серед яких є Kubernetes. Це відкрите програмне забезпечення, що забезпечує управління контейнерами у кластерному середовищі, автоматизуючи їхнє розгортання, масштабування, оновлення та моніторинг [2]. Хоча початково Kubernetes тісно інтегрувався з Docker, з часом архітектура Kubernetes перейшла до універсального інтерфейсу контейнерів (Container Runtime Interface), що дозволило використовувати альтернативні механізми виконання контейнерів (наприклад, containerd або CRI-O).

До складу сучасного технологічного середовища Docker входить також велика кількість додаткових утиліт і сервісів, що охоплюють питання безпеки, автоматизації, управління та спостереження. Серед них:

- Docker Desktop – локальне середовище для розробки з графічним інтерфейсом;
- Docker Swarm – вбудована система оркестрації від Docker;
- Portainer – веб-інтерфейс для управління контейнерами та кластером;

- Harbor – корпоративне сховище образів з функціями авторизації, сканування вразливостей і політик доступу;
- Traefik – динамічний зворотний проксі та маршрутизатор для мікросервісів;
- Інтеграції з CI/CD-системами, такими як GitLab CI, Jenkins, GitHub Actions тощо.

Таким чином, Docker разом із супутніми інструментами формує інтегровану інфраструктуру, що охоплює всі етапи життєвого циклу програмного забезпечення. Завдяки своїй модульності, розширюваності та широкій підтримці професійної спільноти, ця платформа є одним з основних інструментів сучасної DevOps-практики, закладаючи підґрунтя для побудови надійних, автоматизованих і масштабованих програмних рішень.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ІГРОВОГО ВЕБДОДАТКУ З УРАХУВАННЯМ КОНТЕЙНЕРИЗАЦІЇ

2.1. Аналіз вимог та структура застосунку на основі контейнерної архітектури

Під час розробки програмного забезпечення особливо важливим є чітке формулювання вимог до функціоналу та технічних характеристик системи, адже саме це забезпечує узгодженість дій усіх учасників проєкту та дає змогу гарантувати, що кінцевий продукт буде відповідати очікуванням користувачів і замовників. Функціональність системи полягає у здатності виконувати конкретні дії, що задовольняють потреби користувача. Наприклад, користувач повинен мати змогу реєструватися, входити в систему, здійснювати пошук інформації, працювати з базою даних, додавати або змінювати записи, формувати звіти та отримувати відповіді на свої запити [15]. Такі функціональні можливості мають бути визначені точно, логічно, без двозначностей, а також перевірятися на відповідність реальним потребам, не суперечити одна одній і бути зрозумілими для всіх сторін, які залучені до розробки. Їх часто описують за допомогою формулювань, що чітко вказують на обов’язкову поведінку системи в тих чи інших умовах [12].

Разом із функціональними вимогами важливу роль відіграють і технічні характеристики, які визначають, як саме має бути реалізовано кожен аспект роботи системи. Вони охоплюють архітектуру програмного забезпечення, вибір мови програмування, інструментів розробки, структуру зберігання даних, а також визначають вимоги до серверного середовища, рівня безпеки, швидкодії, масштабованості та інтеграційної здатності системи. Наприклад, архітектура повинна дозволяти ефективну взаємодію між окремими модулями, забезпечувати надійність та стабільність роботи програми навіть під високим

навантаженням[12]. Вибір інструментів залежить від їх підтримки, документації, досвіду розробників і можливостей для подальшого розширення. Особливу увагу приділяють безпеці, зокрема реалізації механізмів автентифікації та авторизації, шифруванню даних і захисту від зовнішніх атак[9]. Продуктивність оцінюється за критеріями часу реакції, обробки запитів і здатності працювати з великою кількістю одночасних користувачів без погіршення якості обслуговування.

У контексті об'єктно-орієнтованого програмування (ООП) та розробки якісного програмного забезпечення, важливо дотримуватися принципів SOLID, які сприяють створенню гнучкої, масштабованої та легко підтримуваної архітектури. Принцип єдиної відповідальності передбачає, що кожен клас або модуль повинен мати лише одну причину для змін, тобто відповідати за одну конкретну функцію або завдання. Це сприяє зменшенню складності коду та полегшує його тестування і обслуговування. Принцип відкритості/закритості вказує на те, що програмні сутності повинні бути відкритими для розширення, але закритими для модифікації, що дозволяє додавати нову функціональність без зміни існуючого коду, зменшуючи ризик виникнення помилок. Принцип підстановки Лісков (Liskov Substitution Principle) стверджує, що об'єкти підкласів повинні бути взаємозамінними з об'єктами батьківських класів без порушення коректності програми, забезпечуючи надійність і передбачуваність поведінки системи. Принцип розділення інтерфейсу (Interface Segregation Principle) рекомендує створювати вузькоспеціалізовані інтерфейси, щоб клієнти не були змушені залежати від методів, які вони не використовують, що сприяє зменшенню зв'язності та підвищенню гнучкості системи. Принцип інверсії залежностей (Dependency Inversion Principle) передбачає, що високорівневі модулі не повинні залежати від низькорівневих; обидва повинні залежати від абстракцій, що дозволяє зменшити зв'язність між компонентами та полегшує тестування і модифікацію коду [12].

Таким чином, функціональні вимоги та технічні специфікації тісно взаємопов’язані між собою: перші визначають, що система повинна робити, а другі – як вона має це робити. Для досягнення якісного результату важливо, щоб на кожному етапі розробки проводилася постійна комунікація між замовниками, розробниками, тестувальниками та іншими зацікавленими сторонами. Це дає змогу вчасно коригувати вимоги, адаптуватися до змін у середовищі чи бізнес-логіці проєкту та в результаті створити повноцінну, надійну та зручну у користуванні систему, яка повністю відповідає поставленим цілям.

Під час розробки програмного забезпечення, зокрема гри «Дурак», особливо важливим є чітке формулювання вимог до функціоналу та технічних характеристик системи. Це забезпечує узгодженість дій усіх учасників проєкту та гарантує, що кінцевий продукт відповідатиме очікуванням користувачів і замовників [12]. Функціональність системи полягає у здатності виконувати конкретні дії, що задовольняють потреби користувача. Наприклад, користувач повинен мати змогу реєструватися, входити в систему, здійснювати пошук інформації, працювати з базою даних, додавати або змінювати записи, формувати звіти та отримувати відповіді на свої запити. Такі функціональні можливості мають бути визначені точно, логічно, без двозначностей, а також перевірятися на відповідність реальним потребам, не суперечити одна одній і бути зрозумілими для всіх сторін, які залучені до розробки. Їх часто описують за допомогою формулювань, що чітко вказують на обов’язкову поведінку системи в тих чи інших умовах [15].

Разом із функціональними вимогами важливу роль відіграють і технічні характеристики, які визначають, як саме має бути реалізовано кожен аспект роботи системи. Вони охоплюють архітектуру програмного забезпечення, вибір мови програмування, інструментів розробки, структуру зберігання даних, а також визначають вимоги до серверного середовища, рівня безпеки, швидкодії, масштабованості та інтеграційної здатності системи [15]. Наприклад,

архітектура повинна дозволяти ефективну взаємодію між окремими модулями, забезпечувати надійність та стабільність роботи програми навіть під високим навантаженням. Вибір інструментів залежить від їх підтримки, документації, досвіду розробників і можливостей для подальшого розширення. Особливу увагу приділяють безпеці, зокрема реалізації механізмів автентифікації та авторизації, шифруванню даних і захисту від зовнішніх атак [9]. Продуктивність оцінюється за критеріями часу реакції, обробки запитів і здатності працювати з великою кількістю одночасних користувачів без погіршення якості обслуговування

У сучасній розробці програмного забезпечення використання контейнеризації та мікросервісної архітектури стало стандартом для забезпечення масштабованості, гнучкості та ефективності. У контексті гри було реалізовано три основні компоненти – Frontend, Backend та бази даних – у вигляді окремих контейнерів дозволяє досягти високої модульності та спрощує процес розгортання та підтримки системи [16].

Frontend, реалізовано як окремий контейнер, що відповідає за інтерфейс користувача та взаємодію з гравцем. Цей компонент побудований за допомогою сучасного фреймворка React, що дозволяє створити динамічний та інтерактивний інтерфейс. Frontend взаємодіє з Backend через HTTP-запити, використовуючи методи GET для отримання даних, POST для створення нових записів, PUT для оновлення існуючих та DELETE для видалення [14].

Backend, у свою чергу, реалізує бізнес-логіку гри та обробляє запити від Frontend. Він також розгорнутий у окремому контейнері та побудований на основі Django. Backend забезпечує обробку RESTful API-запитів та підтримує WebSocket-з'єднання для забезпечення реального часу гри між гравцями. Це дозволяє миттєво передавати інформацію про хід гри, карти на руках та інші важливі події.

База даних, PostgreSQL, також розгорнута у окремому контейнері. Вона відповідає за зберігання інформації про користувачів, результати ігор,

статистику та інші дані, необхідні для функціонування гри. Backend взаємодіє з базою даних через відповідні драйвери, забезпечуючи ефективний доступ до даних.

Для координації роботи всіх трьох контейнерів використовується Docker Compose, який дозволяє описати конфігурацію сервісів у файлі docker-compose.yml. Це забезпечує зручне розгортання та масштабування системи. Наприклад, у файлі docker-compose.yml можна визначити залежності між сервісами, порти, змінні середовища та інші параметри, необхідні для коректної роботи системи.

Використання мікросервісної архітектури та контейнеризації дозволяє легко оновлювати окремі компоненти системи без впливу на інші, забезпечує гнучкість у виборі технологій для кожного сервісу та спрощує процес тестування та розгортання [12]. Крім того, така структура сприяє підвищенню надійності та масштабованості системи, що є критично важливим для онлайн-ігор, де важлива стабільна та швидка взаємодія між гравцями.

Таким чином, реалізація гри «Дурак» з використанням окремих контейнерів для Frontend, Backend та бази даних, а також застосування Docker Compose для їх координації, є ефективним підходом, що відповідає сучасним вимогам до розробки масштабованих та надійних вебзастосунків.

2.2. Вибір технологій для Frontend частини вебзастосунку

Frontend – це клієнтська частина вебзастосунку, з якою безпосередньо взаємодіє користувач. Високоякісна реалізація інтерфейсу є критично важливою для забезпечення зручності, інтуїтивної зрозумілості та привабливого візуального представлення цифрових продуктів. Базові технології, що історично використовуються у Frontend-розробці, включають HTML, CSS і JavaScript – кожна з яких виконує унікальну роль і постійно розвивається [20].

HTML (HyperText Markup Language) визначає структуру вебсторінок. Останні оновлення цієї мови значно розширили її можливості у контексті адаптивного дизайну, що є необхідним для підтримки роботи застосунків на широкому спектрі пристроїв – від мобільних телефонів до великих моніторів.

CSS (Cascading Style Sheets) забезпечує стилізацію елементів інтерфейсу, дозволяючи створювати складні анімації, макети й теми, а також реалізовувати принципи респонсивного дизайну.

JavaScript, у свою чергу, еволюціонувала із простої скриптової мови до повнофункціональної мови програмування з підтримкою об'єктно-орієнтованого підходу, асинхронного програмування (через колбеки, проміси та async/await), а також великого набору бібліотек і фреймворків. Це дозволяє реалізовувати клієнтську логіку, інтерактивність і навіть рендеринг сторінок без необхідності повного перезавантаження.

Усі три технології – HTML, CSS і JavaScript – формують основу сучасної Frontend-розробки. На їхній базі створено низку потужних фреймворків, зокрема React, Vue та Angular, які значно спрощують створення масштабованих, ефективних і кросплатформенних вебзастосунків. У поєднанні з технологіями контейнеризації це дозволяє реалізовувати гнучкі архітектури, які відповідають вимогам швидкодії, адаптивності та універсальності.

Особливу роль у сучасному Frontend відіграє фреймворк React, який забезпечує компонентний підхід до розробки. Компоненти в React містять логіку, шаблон і стан (state), а також використовують пропси (props) для передачі даних. Такий підхід дозволяє створювати масштабовані та повторно використовувані інтерфейсні елементи.

Для керування глобальним станом застосовуються бібліотеки Redux, Zustand або Context API. Маршрутизація реалізується через React Router, що дозволяє створювати SPA (Single Page Applications) з навігацією без перезавантаження сторінок.

Для обміну даними між Frontend та Backend широко використовуються HTTP-клієнти, зокрема Axios – для взаємодії з REST API. Бібліотека React Query (хоча вважається застарілою в окремих контекстах) забезпечує високий рівень автоматизації: кешування, фонове оновлення даних, обробку помилок, а також реалізацію механізмів попереднього завантаження (prefetching) та повторних запитів (retry), що покращує користувацький досвід.

React Native є розширенням React, орієнтованим на розробку нативних мобільних застосунків для платформ iOS та Android. Він дозволяє створювати мобільні інтерфейси за допомогою компонентів типу <View>, <Text> тощо, що рендеряться у вигляді нативних елементів. Важливою перевагою React Native є можливість повторного використання значної частини бізнес-логіки між веб- та мобільною версіями, що дозволяє скоротити витрати на розробку. Водночас слід зазначити складність процесу збірки (build) і значне навантаження на систему під час компіляції великих проєктів.

Ще одним важливим фреймворком є Next.js, що побудований на базі React і підтримує серверний рендеринг (SSR), статичну генерацію (SSG), динамічний імпорт, API-роути та повну інтеграцію з TypeScript [10]. Next.js забезпечує покращену швидкість завантаження сторінок, автоматичне розбиття коду (code splitting), SEO-оптимізацію та зручну файл-орієнтовану маршрутизацію. Крім того, Next.js дозволяє реалізовувати повноцінні Frontend-Backend рішення в межах одного стеку технологій.

У сучасній веброзробці, де динамічність і зручність інтерфейсу користувача відіграють критичну роль, ефективна та надійна взаємодія з сервером є не лише бажаною, а й необхідною умовою. Значна частина сучасних вебзастосунків базується на використанні REST або GraphQL API, що зумовлює потребу в інструментах, які спрощують і автоматизують процес обміну даними між клієнтською та серверною частинами. У контексті розробки на базі React, найпоширенішими засобами реалізації таких задач є бібліотеки Axios та React Query, що отримали широке визнання у професійному середовищі.

Axios є популярним HTTP-клієнтом, який забезпечує простий та гнучкий інтерфейс для здійснення запитів до API. Його основними перевагами є підтримка конфігурованих запитів, зокрема додавання токенів автентифікації, налаштування інтерцепторів для обробки помилок, а також можливість гнучкого управління заголовками та тілами запитів. Axios є особливо корисним у випадках, коли розробник потребує повного контролю над логікою запитів або коли взаємодія з backend-частиною має специфічні особливості.

Натомість React Query орієнтована на автоматизацію обробки даних, отриманих з API. Завдяки інтеграції з React через спеціальні хуки (наприклад, useQuery, useMutation), ця бібліотека значно спрощує реалізацію складних сценаріїв роботи з асинхронними даними. React Query автоматизує кешування, оновлення даних у фоновому режимі, обробку помилок та реалізацію таких функцій, як попереднє завантаження даних (prefetching), повторні запити при помилках (retry), пагінація, нескінченне прокручування (infinite scroll) та оптимістичне оновлення стану. Усе це сприяє підвищенню якості користувацького досвіду та зменшенню навантаження на розробника.

Завдяки зазначеним бібліотекам розробники отримують можливість обирати між ручним та автоматизованим підходом до обробки API-запитів, залежно від складності та специфіки проєкту. Обидва підходи є ефективними інструментами у побудові сучасних інтерфейсів на основі React.

Крім REST-орієнтованої взаємодії, усе більшої популярності набувають технології, орієнтовані на реальний час, де критичним фактором є мінімізація затримок у передаванні даних. У цьому контексті ключову роль відіграє протокол WebSocket, який забезпечує встановлення постійного двонаправленого TCP-з’єднання між клієнтом і сервером. На відміну від традиційних HTTP-запитів, які потребують багаторазових з’єднань, WebSocket дозволяє обмінюватися повідомленнями без додаткових витрат на встановлення кожного запиту, що суттєво знижує затримки та забезпечує майже миттєву реакцію інтерфейсу [10].

У середовищі React реалізація WebSocket може бути виконана як за допомогою вбудованого API, так і за допомогою спеціалізованих бібліотек, зокрема Socket.IO, яка пропонує додаткову функціональність: автоматичне повторне з'єднання, підтримку каналів зв'язку (rooms), обробку подій та зручну API для передачі даних. Інтеграція WebSocket у React-додатки зазвичай реалізується через кастомні хуки, які дозволяють ефективно керувати станом підключення протягом усього життєвого циклу компонента [7].

Особливої ефективності вдається досягти при комбінації WebSocket з бібліотеками керування глобальним станом, як Redux або Context API, що дозволяє синхронізувати дані, що надходять у реальному часі, з відповідними інтерфейсами користувача. Такий підхід забезпечує узгодженість, стабільність та масштабованість навіть у складних розподілених системах.

Таким чином, впровадження WebSocket у поєднанні з сучасними бібліотеками для роботи з API значно розширює можливості фронтенд-розробника, дозволяючи створювати високореактивні, масштабовані та зручні для користувача вебзастосунки. У застосунках, які потребують миттєвої реакції – чатах, фінансових дашбордах, онлайн-іграх – використання таких технологій стає стандартом, а не винятком.

2.3. Вибір технологій для Backend частини вебзастосунку

Вибір серверної технології є ключовим етапом у процесі створення вебдодатку, оскільки саме Backend відповідає за бізнес-логіку, збереження даних та обмін інформацією з клієнтською частиною. Серед наявних рішень особливо популярною є мова програмування Python разом із фреймворком Django, які поєднують простоту, гнучкість і функціональну повноту.

Python вирізняється лаконічним і зрозумілим синтаксисом, що сприяє швидкій розробці та підтримці коду. Його великий набір бібліотек і активна

спільнота створюють сприятливі умови для реалізації найрізноманітніших задач – від обробки даних до побудови API [9].

Django є високорівневим фреймворком для розробки вебзастосунків на мові програмування Python, який надає цілісну систему засобів для побудови серверної частини сучасних вебпорталів. Його функціональні можливості охоплюють маршрутизацію, об’єктно-реляційне відображення (ORM) для роботи з базами даних, механізми автентифікації та авторизації, вбудовану адміністративну панель, а також модулі забезпечення безпеки від типових загроз (таких як XSS, CSRF, SQL-ін’єкції тощо). Такий підхід дозволяє значно скоротити час розробки, зосередившись на реалізації бізнес-логіки, а не на вирішенні рутинних технічних завдань.

У порівнянні з іншими поширеними фреймворками, такими як Flask (Python), Express.js (Node.js) та Ruby on Rails (Ruby), фреймворк Django демонструє вищий рівень структурованості, архітектурної впорядкованості та функціональної завершеності. Його основною ідеологією є принцип «batteries included», що означає наявність широкого набору вбудованих рішень, які охоплюють ключові аспекти веброзробки – від аутентифікації та адміністрування до маршрутизації, ORM та реалізації політик безпеки. На відміну від Flask чи Express, які надають лише мінімальний каркас і потребують додаткової інтеграції сторонніх бібліотек для базової функціональності, Django вже з самого початку постачає повноцінний набір інструментів, придатних для реалізації як невеликих, так і масштабованих систем.

Це забезпечує значні переваги для великих проєктів з чітко визначеними вимогами до архітектури, безпеки, стандартизації процесів та довгострокової підтримки. Зокрема, наявність внутрішньо узгоджених компонентів мінімізує ризики конфліктів між бібліотеками та пришвидшує початкову фазу розробки, скорочуючи витрати на інтеграцію, налаштування та тестування сторонніх рішень. Крім того, Django забезпечує суворе дотримання принципів Model-View-Template (MVT), що сприяє чіткому розмежуванню логіки даних,

представлення й контролю, полегшуючи масштабування та обслуговування застосунків.

Завдяки детальній офіційній документації, широкій екосистемі додаткових пакетів, гнучкій системі розширення та активній міжнародній спільноті розробників, Django залишається одним із найбільш ефективних засобів для створення вебзастосунків із підвищеними вимогами до безпеки, масштабованості та підтримуваності [9]. Ці властивості забезпечують стабільність розробки як на етапі створення MVP, так і при подальшій еволюції програмного продукту до рівня повнофункціональної, розподіленої системи. Цьому Django вважається оптимальним вибором для реалізації систем зі складною бізнес-логікою, підвищеними вимогами до безпеки, а також високим навантаженням або перспективами подальшого розширення.

Ефективне управління даними є основою функціональності будь-якого сучасного вебдодатку. У цьому контексті реляційна база даних PostgreSQL займає провідне місце завдяки своїй надійності, масштабованості та гнучкості. Вона підтримує складні SQL-запити, транзакції, індексацію, розширення та високий рівень відповідності стандартам ACID, що гарантує цілісність і узгодженість даних.

PostgreSQL добре інтегрується з більшістю Backend, зокрема з Django, який має потужну ORM-систему для зручної взаємодії з базою без написання сирого SQL. Така інтеграція дозволяє легко створювати та мігрувати структури таблиць, здійснювати зв’язки між сутностями, виконувати фільтрацію, агрегацію та інші запити високого рівня.

Завдяки відкритому коду, активній спільноті та широкому набору розширень PostgreSQL ефективно використовується як у невеликих проєктах, так і в високонавантажених системах, що потребують стійкості до збоїв, реплікації, шардінгу чи аналітичної обробки даних [11]. Це універсальне рішення, що поєднує потужність традиційних СУБД з можливостями сучасної розробки.

Для реалізації функціоналу в реальному часі, такого як чати, сповіщення чи оновлення даних без перезавантаження сторінки, використовується WebSocket – протокол, який забезпечує постійне двостороннє з'єднання між клієнтом і сервером.

Django Channels розширює можливості класичного Django, додаючи підтримку асинхронної обробки запитів та WebSocket-з'єднань. Ця технологія дозволяє створювати масштабовані асинхронні застосунки, які обробляють не лише HTTP, а й протоколи реального часу. Channels базується на ASGI – сучасному інтерфейсі між вебсервером і фреймворком, що забезпечує асинхронність у Python-застосунках. Завдяки цьому Django може працювати з подієвими циклами, чергами повідомлень і брокерами, як Redis, для керування підключеннями та розповсюдженням повідомлень [18].

Інтеграція з Django Channels відкриває можливості створення інтерактивного користувацького досвіду, де події на сервері одразу відображаються у Frontend. Це особливо актуально для ігор, бірж, чатів або будь-яких систем із високою динамікою даних. Такий підхід дозволяє ефективно поєднувати класичну синхронну архітектуру Django з сучасними вимогами до асинхронної взаємодії в реальному часі [10].

Frontend відіграє ключову роль у формуванні користувацького досвіду, адже саме він відповідає за візуальну складову, зручність навігації та інтерактивність інтерфейсу. Основу інтерфейсу складають компоненти, які є повторно використовуваними елементами з власною логікою та стилями. Їхній стан контролюється внутрішніми механізмами React, що дозволяє оперативно реагувати на дії користувача та змінювати інтерфейс без перезавантаження сторінки [7].

Взаємодія з користувачем реалізується через події (onClick, onChange тощо), які викликають відповідні функції та змінюють стан додатка. Для складніших сценаріїв застосовується керування глобальним станом, а також взаємодія з API – завдяки асинхронним запитам компоненти отримують дані та

оновлюються динамічно [14]. Цей підхід забезпечує високий рівень інтерактивності та чуйності інтерфейсу.

Особлива увага приділяється адаптивності дизайну, що дозволяє інтерфейсу коректно відображатися на різних пристроях. Комбінація React з сучасними CSS-інструментами (Tailwind, CSS Modules або Styled Components) дозволяє ефективно стилізувати компоненти та підтримувати зручну структуру коду. У результаті формується зрозумілий, привабливий і функціональний інтерфейс, що підтримує комфортну та логічну взаємодію користувача із застосунком.

Реалізація взаємодії в режимі реального часу є критично важливою для онлайн-ігор, зокрема карткових, де гравці мають миттєво бачити дії один одного. Для цього застосовується протокол WebSocket, який забезпечує постійне двостороннє з'єднання між клієнтом і сервером без потреби повторного встановлення з'єднання для кожного запиту.

РОЗДІЛ 3
ТЕХНІЧНА РЕАЛІЗАЦІЯ ІГРОВОГО ВЕБДОДАТКУ

3.1. Налаштування інфраструктури розробки з використанням контейнерних технологій

Першим етапом розробки гри «Дурак Онлайн» стало налаштування середовища, яке забезпечує комфортну та ефективну роботу над проєктом. Для локального запуску було обрано стек технологій, що добре піддається контейнеризації: React та PixiJS для Frontend, Python та Django для Backend, PostgreSQL для бази даних [11].

Розробка відбувалася в середовищі Visual Studio Code з використанням розширень Docker, Prettier, ESLint, Python та React. Це дозволило підвищити продуктивність і забезпечити єдиний стиль кодування.

```
FROM python:3.11-alpine
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apk update && apk add --no-cache \
    gcc musl-dev postgresql-dev python3-dev libffi-dev \
    cargo jpeg-dev zlib-dev
WORKDIR /app

RUN pip install --upgrade pip && pip install \
    "Django>=4.0" \
    daphne \
    channels \
    channels_redis \
    psycopg2-binary \
    django-cors-headers \
    django-environ \
    python-decouple \
    djangorestframework \
    requests
COPY . .

EXPOSE 8000

CMD ["sh", "-c", "python manage.py migrate && daphne -b 0.0.0.0 -p 8000 cerds_game.asgi:application"]
```

Рисунок 3.1. Dockerfile для Backend

Для створення ізольованих середовищ розробки було встановлено Docker Desktop (див рисунок 3.3). Усі необхідні залежності проєкту були описані у відповідних конфігураційних файлах таких як docker-compose.yml – для об’єднання сервісів у спільну мережу (див. рисунок 3.2)

```yaml
services:
  web:
    build: .
    command: sh -c "python manage.py migrate && daphne -b 0.0.0.0 -p 8000 cerds_game.asgi:application"
    volumes:
      - .:/cerds_game
    ports:
      - "8000:8000"
    env_file:
      - .env
    depends_on:
      - db
      - redis

  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
  redis:
    image: redis:7
    ports:
      - "6379:6379"
volumes:
  postgres_data:
```

Рисунок 3.2. docker-compose.yml для Backend сторони

Кожен компонент додатку має свою окрему директорію та налаштування, що дозволяє масштабувати проєкт та легко знаходити проблеми у конкретних частинах системи.

Після підготовки середовища команда розробки змогла швидко запустити додаток у контейнерах, що дозволило одразу перевіряти працездатність взаємодії компонентів і уникнути типових проблем з налаштуванням залежностей.

Контейнеризація основних компонентів вебгри «Дурак Онлайн» дозволила досягти стабільної роботи та легкого розгортання проєкту на будь-

якому середовищі. Кожна частина додатку – Frontend, Backend та база даних – була упакована в окремий контейнер, що забезпечує чітку ізоляцію, спрощене управління залежностями та гнучкість при масштабуванні.

Frontend, реалізований за допомогою React та PixiJS, був зібраний у окремому Docker-контейнері з використанням Node.js (див. рисунок 3.4). Це дозволяє швидко будувати статичні файли для продакшн-режиму та автоматично запускати development-сервер під час розробки.

Backend, побудований на Django, також має власний Dockerfile. Для нього створене окреме середовище, яке містить лише необхідні бібліотеки Python, визначені у requirements.txt. Django відповідає за автентифікацію, бізнес-логіку гри, взаємодію з базою даних та WebSocket-підключення через Django Channels.

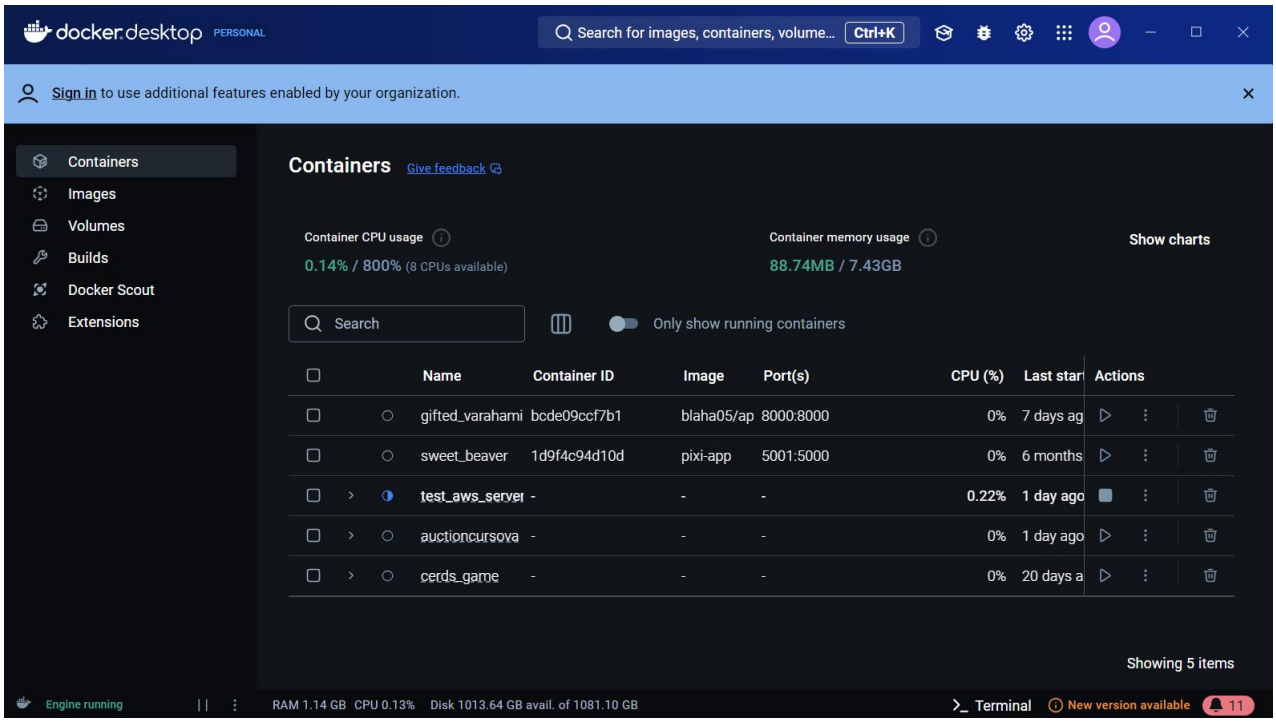


Рисунок 3.3. Демонстрація головного вікна в Docer Desctop

Для зберігання ігрових даних використовується база даних PostgreSQL, що також запускається у вигляді контейнера. Всі сервіси об’єднані в єдину

мережу за допомогою docker-compose, що дозволяє їм взаємодіяти один з одним через імена контейнерів замість IP-адрес [16].

Весь стек контейнерів був протестований на локальному середовищі та легко адаптується до хмарних платформ, таких як Railway чи Render [1].

Завдяки чітко структурованій контейнеризації, додаток став більш надійним, легким для обслуговування та готовим до розгортання в продакшн-середовищах.

```yaml
version: '3.8'

services:
  frontend:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:80"
    container_name: react-pixi-frontend
    restart: always
```

Рисунок 3.4. Файл docker-compose.yml для Frontend сторони

3.2. Створення та налаштування бази даних PostgreSQL у контейнері

У рамках розробки вебдодатку було прийнято рішення використовувати реляційну базу даних PostgreSQL, що забезпечує надійне збереження інформації про користувачів, ігрові сесії, карти та хід гри. Задля забезпечення ізоляції, простоти масштабування та легкого розгортання, база даних була контейнеризована за допомогою Docker.

Налаштування бази даних відбувається за допомогою файлу конфігурації docker-compose.yml, у якому описано сервіс db, що базується на офіційному

образі postgres:15. Також було використано volume для збереження даних між перезапусками:

```yaml
services:
  db:
    image: postgres:15
    container_name: durok_postgres
    restart: always
    environment:
      POSTGRES_DB: durokdb
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret123
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - backend_network

volumes:
  postgres_data:
```

Рисунок 3.5. Dockerfile для бази данних

Налаштування параметрів підключення до бази даних у Django здійснюється через файл settings.py, де дані зчитуються із змінних середовища, що визначені в окремому .env файлі:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRES_DB'),
        'USER': os.getenv('POSTGRES_USER'),
        'PASSWORD': os.getenv('POSTGRES_PASSWORD'),
        'HOST': 'db',
        'PORT': '5432',
    }
}
```

Після запуску контейнерів та виконання міграцій у Django створюються необхідні таблиці у базі даних. У процесі розробки для перевірки вмісту БД.

Контейнеризація бази даних дозволяє досягти наступних переваг:

- швидке розгортання бази на будь-якому середовищі без додаткової конфігурації;
- відокремлення даних від прикладної логіки;
- зручність у тестуванні та відновленні;
- гнучке управління мережею та безпекою.

Для забезпечення роботи ігрового вебзастосунку було спроєктовано реляційну базу даних на основі PostgreSQL. Було створено кілька основних моделей, які відображають логіку гри та взаємодію користувачів:

- Room (Кімната) – модель, яка відповідає за ігрову сесію. Вона зберігає інформацію про кількість учасників, ігрову колоду, ставки, стан гри (атака, передача ходу, шахрайство тощо), порядок гравців і список карт.
- Player (Гравець) – модель, що описує конкретного учасника гри. Вона містить ім’я, унікальний ідентифікатор, стан готовності, чергу ходу, належність до кімнати.

Глобальна модель Player – окрема модель, яка зберігає глобальну інформацію про користувача незалежно від кімнати: його унікальний user_id, кількість виграних призів, а також функціонал друзів (реалізовано через зв’язок many-to-many).

Взаємозв’язки між моделями побудовані за принципом «один до багатьох»: одна кімната може мати багато гравців (ForeignKey), а також «багато до багатьох» між гравцями для реалізації друзів.

Переваги використання PostgreSQL у даному проєкті:

- JSONField дозволяє зберігати динамічні списки карт, порядок гравців тощо без необхідності створювати додаткові таблиці.
- Транзакційна цілісність гарантує коректність даних під час оновлення стану гри.

- Потужна система індексації та підтримка складних запитів роблять PostgreSQL оптимальним вибором для високонавантажених вебзастосунків.
- Контейнеризація забезпечує ізольованість БД та швидке розгортання як у локальному середовищі, так і на хмарних платформах.

Приклад ER-діаграми:

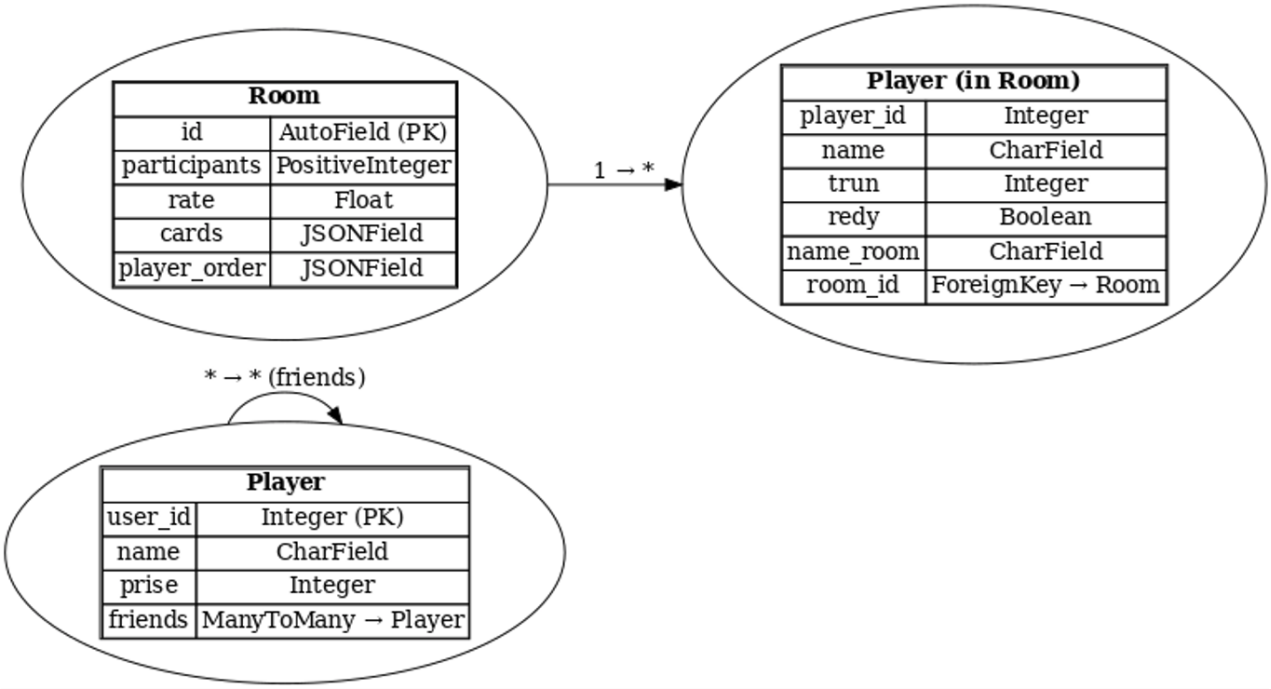


Рисунок 3.6. Зв’язок між таблицями базами даних

Завдяки такій структурі, база даних підтримує гнучку та масштабовану логіку гри, дозволяючи як багатокористувацькі сесії, так і збереження глобального профілю гравця.

3.3. Використання WebSocket для реалізації ігрової логіки з розподілом функціоналу по сервісах

Щоб багатокористувацькі вебдодатки працювали в реальному часі, критично важливо мати ефективний спосіб обміну подіями між усіма підключеними клієнтами. У контексті Django та Django Channels, Redis

виступає в ролі брокера повідомлень (message broker), що є ключовим для забезпечення безперебійної взаємодії через WebSocket-з'єднання .

Redis, або Remote Dictionary Server, є надзвичайно швидким сховищем даних у пам'яті, що працює за принципом ключ-значення. Саме ця швидкість робить Redis ідеальним вибором для WebSocket-з'єднань, адже там кожне повідомлення має бути доставлене та отримане миттєво.

У світі Django Channels, Redis слугує так званим «channel layer» – це такий собі проміжний шар, що дає змогу передавати повідомлення не лише між різними процесами, а й навіть між окремими серверами. Це стає незамінним, коли, наприклад, декілька гравців знаходяться в одній ігровій кімнаті, і будь-які оновлення – чи то нова карта, чи хід іншого гравця – мають миттєво з'являтися у всіх учасників .

Для інтеграції Redis з Django Channels у settings.py додається наступна конфігурація:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("localhost", 6379)],
        },
    },
}
```

Redis також контейнеризується у межах docker-compose.yml, що дозволяє легко розгортати середовище для тестування та продакшну:

```
redis:
  image: redis:7
  ports:
    - "6379:6379"
```

Таким чином, Redis забезпечує надійний механізм для обміну подіями в реальному часі між клієнтами гри через WebSocket, дозволяючи досягти високої швидкодії, масштабованості та узгодженості стану гри.

Для забезпечення інтерактивної багатокористувацької взаємодії в ігровому вебдодатку було реалізовано обробку подій через WebSocket на базі Django Channels. Основним компонентом є клас RyoomCardsGame, який відповідає за підключення користувачів, прийом і відправку повідомлень, а також керування станом гри.

```
import json
from channels.generic.websocket import AsyncWebsocketConsumer
from channels.db import database_sync_to_async
from .service import add_player_room, remove_player_room, redy, check_redy, add_goat, add_cards_to_player, send_to_others, trun_teka, trun_whipped, to_choose_cards, swap_player_room, trun_pass, check_to_win, restart
from .models import Room, Player

class RyoomCardsGame(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_id = self.scope['url_route']['kwargs']['room_id']
        self.number_players = self.scope['url_route']['kwargs']['number_players']

        # Перевірка на наявність кімнати
        if not await self.room_exists(self.room_id):
            await self.close()
            return
        self.room = await self.get_room(self.room_id)
        self.group_name = f"group_{self.room_id}"
        await self.accept()
        await self.channel_layer.group_add(self.group_name, self.channel_name)
        await self.room.ensure_cards()

    async def receive(self, text_data):
        json_data = json.loads(text_data)
        action = json_data.get('type')

        if action == 'user':
            await add_player_room(self, json_data, self.room_id, self.number_players)
        if action == 'swap':
```

```
            await swap_player_room(self, self.room_id, json_data.get('number'))
        if action == 'redy':
            # await self.delete_all_players()
            await redy(self, self.room_id)
            if await check_redy(self, self.room_id):
                await add_goat(self, self.room_id)
                await add_cards_to_player(self, self.room_id) # end create to trum
        if action == 'audit':
            await to_choose_cards(self, self.room_id, json_data)
        if action == 'teka':
            await trun_teka(self, self.room_id)
            await send_to_others(self, self.room_id, json_data)
        if action == 'whipped':
            await trun_whipped(self, self.room_id, self.number_players)
            # await send_to_others(self, self.room_id, json_data)
        if action == 'pass':
            await trun_pass(self, self.room_id, self.number_players)
        if action == 'def' or action == 'atack':
            await send_to_others(self, self.room_id, json_data)
        if action == 'win':
            await check_to_win(self, self.room_id, json_data.get('id'))
        if action == 'restart':
            await restart(self, self.room_id, self.number_players)
    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(self.group_name, self.channel_name)
        await remove_player_room(self, self.room_id)
        await self.remove_player_from_db()
        await self.remove_player_and_check_room(self.room_id)

    @database_sync_to_async
    def remove_player_and_check_room(self, room_id):
        room = Room.objects.get(id=room_id)
        if not room.players.exists():
            room.player_order.clear()
            room.delete()

    @database_sync_to_async
```

```
def remove_player_from_db(self):
    player = Player.objects.get(name_room=self.channel_name)
    player.delete()

@database_sync_to_async
def get_or_create_room(self, room_id):
    room, created = Room.objects.get_or_create(id=room_id)
    return room

@database_sync_to_async
def delete_all_players(self):
    Player.objects.all().delete()
    self.room.player_order = []  # Присвоюємо порожній список
    self.room.save()  # Зберігаємо зміни до бази даних

@database_sync_to_async
def get_room(self, room_id):
    return Room.objects.get(id=room_id)

@database_sync_to_async
def room_exists(self, room_id):
    return Room.objects.filter(id=room_id).exists()

async def message(self, event):
    message = event['message']
    await self.send(text_data=json.dumps({
        'message': message
    }))
```

Коли користувач підключається, ми спочатку перевіряємо, чи існує потрібна ігрова кімната (room_id). Якщо так, клієнт одразу приєднується до WebSocket групи повідомлень для цієї кімнати. Це забезпечує миттєву трансляцію всіх ігрових подій до кожного учасника.

Серце обробки подій знаходиться в методі receive. Він отримує JSON-повідомлення від клієнтів, розпізнає тип дії (type) і потім передає завдання відповідним асинхронним сервісам. Ці сервіси винесені в окремий модуль, що

називається service. Такий розподіл значно покращує читабельність та легкість підтримки коду, адже вся ігрова логіка тепер чітко розділена на окремі, зрозумілі функції: від додавання гравця до кімнати та перевірки готовності учасників, до роздачі карт, обробки ходів та інших ігрових взаємодій .

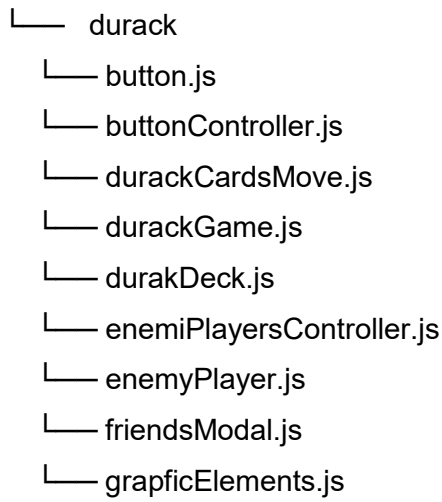
Таке відокремлення бізнес-логіки в окремі сервіси не тільки спрощує тестування кожного компонента, але й робить додаток набагато легшим для масштабування.

Коли користувач відключається, його одразу видаляють як з кімнати, так і з бази даних. Якщо ж після цього кімната залишається порожньою, її також автоматично видаляють, щоб уникнути накопичення застарілих даних.

Для безпечної взаємодії з базою даних в асинхронному середовищі WebSocket-консьюмера ми використовуємо декоратор @database_sync_to_async. Він дозволяє нам безпечно викликати ORM-методи, не порушуючи асинхронний потік.

3.4. Структура клієнтської частини вебдодатку

Розроблена гра «Дурак» має модульну архітектуру, де кожен модуль відповідає за певний аспект функціональності гри. Такий підхід дозволяє розділити логіку гри на зрозумілі частини, спростити підтримку та масштабування проєкту.



```
└── main.js
└── movePlayers.js
└── timerBar.js
└── wsRoomDurack.js
```

- Button.js містить реалізацію окремої кнопки інтерфейсу, яка реагує на дії користувача.
- ButtonController.js відповідає за управління всіма кнопками в грі, обробляє події кліку та змінює їхній стан.
- DurackCardsMove.js реалізує логіку переміщення карт між ігровими зонами, включаючи анімації.
- DurackGame.js – центральний модуль з основною ігровою логікою, що керує правилами, ходами та станом гри.
- DurakDeck.js управляє колодою карт, їх тасуванням та роздачею гравцям.
- EnemiPlayersController.js відповідає за логіку керування штучними супротивниками.
- EnemyPlayer.js описує поведінку окремого ворожого гравця.
- FriendsModal.js забезпечує відображення модального вікна для соціальної взаємодії
- GrapficElements.js містить компоненти інтерфейсу та візуальні елементи гри.
- Main.js ініціалізує гру та запускає основні процеси.
- MovePlayers.js реалізує передачу ходу між гравцями.
- TimerBar.js відображає таймер для контролю часу ходів.
- WsRoomDurack.js забезпечує роботу WebSocket-з’єднання для обміну подіями між гравцями у реальному часі.

Візуальна частина гри створена за допомогою Canvas, який ми гармонійно вбудували в React-компоненти. Це дозволяє нам легко відображати динамічну графіку (див. рисунок 3.7.), плавні анімації карт та всі інтерактивні елементи інтерфейсу. Об'єднавши React і Canvas, ми отримали сучасний підхід до

створення браузерних ігор, що вирізняється високою продуктивністю та продуманою структурою.

Окрім базового геймплею, ми також передбачили в грі підтримку світлої та темної теми, аби кожен гравець міг обрати найбільш комфортне для себе оформлення. До того ж, перед початком гри користувач має змогу вибрати один із запропонованих ігрових столів, що додає грі елемент персоналізації.

Рисунок 3.7. Демонстрація ігрового стола залежно від теми

РОЗДІЛ 4
РОЗГОРТАННЯ ТА ТЕСТУВАННЯ КЛІЄНТ-СЕРВЕРНОЇ ГРИ

4.1. Вибір хмарної інфраструктури для ігрового вебзастосунку

У процесі розробки ігрового проєкту постало ключове завдання – обрати надійну хмарну, яка б відповідала технічним критеріям проєкту, зокрема: підтримка контейнеризованого розгортання, можливість масштабування, ефективна робота з мережею, інтеграція з іншими сервісами, а також доступна цінова політика, що відповідала б обмеженому бюджету.

Під час аналізу було розглянуто широкий спектр хмарних платформ, від провідних індустріальних рішень (Amazon Web Services, Google Cloud Platform, Microsoft Azure) до більш орієнтованих на розробників платформ (Render, Railway, Vercel, Firebase). Кожен з варіантів мав власні переваги та обмеження. Зокрема, AWS забезпечував високу гнучкість, можливість детального конфігурування інфраструктури, потужні засоби масштабування та повну підтримку Docker. Водночас, цей варіант вимагав глибокого розуміння архітектури й акуратного керування ресурсами, щоб уникнути зайвих витрат.

Google Cloud Platform виявився досить ефективним з точки зору інтеграції з Firebase та наявності зручної системи білінгу, однак вимагав особливої уваги до налаштувань прав доступу. Microsoft Azure, хоч і демонструє високу надійність у корпоративному середовищі, виявився занадто дорогим для невеликих індивідуальних проєктів. Серед платформ, орієнтованих на простоту використання, Heroku вирізнявся інтуїтивністю інтерфейсу та швидкістю розгортання, однак показав обмежену продуктивність, особливо в умовах «холодного старту» та при інтенсивному використанні WebSocket-підключень.

Після детального аналізу доступних хмарних рішень було зосереджено увагу на двох платформах, які найбільш відповідали вимогам проєкту: Railway та Amazon Web Services (AWS).

Платформа Railway продемонструвала високу зручність для розробників, завдяки простоті розгортання сервісів та інтуїтивному інтерфейсу. Вона забезпечує швидкий запуск Backend-сервісів без потреби в додатковому налаштуванні, що є особливо корисним на етапі прототипування. Зокрема, розгортання бази даних PostgreSQL відбувалося миттєво, без необхідності втручання в ручному режимі. До переваг Railway також можна віднести зручне керування змінними середовища, доступ до журналів подій (логів), функціональність резервного копіювання, а також підтримку автоматизованого розгортання (CI/CD) з GitHub і Docker-інтеграцією. Ці фактори суттєво спростили інтеграцію бази даних із основним ігровим сервером.

Основну серверну логіку було вирішено реалізувати з використанням AWS, зокрема сервісу Amazon ECS (Elastic Container Service), який дозволяє запускати серверні компоненти у Docker-контейнерах з повним контролем над середовищем виконання. Для кешування ігрових даних застосовувався Amazon ElastiCache з Redis, а для забезпечення масштабованості – інструменти автоматичного балансування навантаження. Незважаючи на дещо складніше первинне налаштування у порівнянні з Railway, AWS компенсує це високою гнучкістю, надійністю та широкими можливостями інтеграції з іншими сервісами в рамках єдиної платформи.

Порівнюючи обидві платформи, Railway можна охарактеризувати як ефективне рішення для швидкого розгортання, тестування та обслуговування невеликих сервісів або баз даних. Натомість AWS є потужним інструментом для реалізації повноцінної інфраструктури продакшн-рівня, з акцентом на масштабованість, стабільність і безпеку.

У результаті було обрано гібридну архітектуру, що поєднує Railway та AWS. Для керованої бази даних PostgreSQL використано Railway, завдяки її безкоштовному тарифу, зручному інтерфейсу та швидкому розгортанню. Основна серверна частина була реалізована за допомогою AWS ECS у межах приватної VPC-мережі, з використанням Amazon ElastiCache для Redis. Такий

підхід забезпечив оптимальний баланс між простотою налаштування та потужністю інфраструктури, що дозволило досягти високої ефективності в реалізації архітектури серверної частини гри.

4.2. Тестування гри в умовах продакшн-середовища на хмарній платформі

Контейнеризація програмного забезпечення, зокрема використання Docker для ігрового застосунку «Дурак Онлайн», відкриває широкі можливості для масштабування, гнучкого керування інфраструктурою та забезпечення стабільної роботи системи. Архітектурна модель, в якій кожен компонент (ігровий сервер, клієнтська частина, база даних, кеш-сервіс) розгортається у відокремленому ізольованому контейнері, дозволяє досягти високого рівня модульності та незалежності частин системи. Такий підхід значно спрощує процес масштабування та впровадження нових функцій.

Однією з ключових переваг обраної архітектури є можливість горизонтального масштабування. У разі збільшення навантаження (наприклад, через зростання кількості одночасних користувачів), система дозволяє запускати додаткові копії ігрового сервера, розподіляючи трафік між ними за допомогою балансувальників навантаження, таких як Nginx або AWS Elastic Load Balancer (ELB). Це дає змогу ефективно обслуговувати велику кількість підключень без втрати продуктивності та зменшення якості користувацького досвіду.

Для управління інфраструктурою було використано інструменти Docker Compose та Kubernetes, що надали можливість централізованого контролю над життєвим циклом контейнерів: розгортання, масштабування, оновлення версій, автоматичне відновлення після збоїв тощо. Такий підхід сприяє підвищенню надійності сервісу та зменшенню часу простою, що є критично важливим для онлайн-ігор.

Експериментальне розгортання гри здійснювалося на бюджетному віртуальному сервері AWS EC2. У процесі тестування було виявлено, що при одночасному підключенні кількох користувачів (3-4 гравці) спостерігалися незначні затримки, переважно під час передавання ходу або завантаження графічних елементів. Це пояснюється обмеженістю обчислювальних ресурсів обраного Instance. Незважаючи на це, загальна працездатність гри залишалася задовільною, а всі основні функції виконувалися коректно.

Попри те, що не всі початкові плани вдалося реалізувати повною мірою, головною метою дослідження залишалося вивчення роботи ігрового застосунку «Дурак Онлайн» у хмарному середовищі. Зокрема, було зосереджено увагу на тестуванні стабільності роботи Backend-сервісу під навантаженням: перевірялася ефективність обробки підключень через WebSocket, синхронізація подій між користувачами, загальна продуктивність системи та якість її відгуку у реальному часі.

На основі проведених експериментів та аналізу було сформульовано низку потенційних напрямів розвитку ігрового застосунку. Серед них:

підтримка великої кількості паралельних ігрових сесій (кількасот ігрових кімнат);

зберігання історії ігор та статистичних даних користувачів;

реалізація авторизації через зовнішні сервіси (OAuth, Google/Facebook/Apple ID тощо);

інтеграція з платіжними системами та впровадження механізмів монетизації.

В цілому, застосування контейнеризованої архітектури у поєднанні з обраною хмарною інфраструктурою дозволило створити гнучку, масштабовану та стабільну платформу для розгортання гри. Цей підхід не лише забезпечує високу надійність у поточному стані системи, але й формує стійке підґрунтя для її подальшого функціонального та архітектурного розширення відповідно до зростання вимог проєкту.

На основі проведеного тестування було зроблено висновок, що навіть при обмежених ресурсах хмарна інфраструктура здатна забезпечити належний рівень продуктивності для підтримки повноцінного онлайн-застосунку. Крім того, у разі потреби, масштабування можливе як шляхом запуску додаткових контейнерів, так і за допомогою переходу на потужніші конфігурації Instance. Така гнучкість є однією з головних переваг сучасних хмарних технологій.

ВИСНОВКИ

У кваліфікаційній роботі розглянуто питання розробки клієнт-серверного ігрового вебзастосунку з використанням сучасних технологій контейнеризації та хмарних сервісів. У межах дослідження було проаналізовано історичний розвиток концепції контейнеризації, її переваги порівняно з традиційними підходами в розгортанні програмного забезпечення, а також особливості застосування в реальних проєктах. Окрему увагу приділено порівнянню віртуальних машин і контейнерів, що дало змогу обґрунтувати доцільність використання саме контейнерного підходу як більш ефективного для створення ізольованих, масштабованих і керованих середовищ виконання.

У процесі реалізації проєкту було спроєктовано архітектуру вебзастосунку, орієнтовану на гнучкість, зручність супроводу та подальше масштабування. Клієнтська частина реалізована із застосуванням технологічного стека React та PixiJS, що забезпечило високий рівень інтерактивності та продуктивності користувацького інтерфейсу. Серверна частина створена на базі фреймворку Django з використанням WebSocket-протоколу для підтримки взаємодії між гравцями в режимі реального часу. Серед реалізованих функцій – підтримка тем оформлення (світла та темна тема), можливість вибору ігрового столу, що значно підвищує зручність і привабливість інтерфейсу для кінцевого користувача.

Особливий акцент у роботі зроблено на використанні хмарних платформ для розгортання застосунку. Проведено порівняльний аналіз можливостей найбільш поширених хмарних сервісів, зокрема Google Cloud Platform, Microsoft Azure, Amazon Web Services, Render, Railway, Firebase та Vercel. У процесі вибору платформи враховувалися такі чинники, як підтримка CI/CD, доступність Docker-контейнерів, можливості масштабування, безпека, стабільність роботи під навантаженням, а також цінова політика.

У результаті дослідження було обрано гібридну архітектуру, в межах якої база даних PostgreSQL розгорнута на платформі Railway, яка продемонструвала швидке налаштування, зручний інтерфейс та автоматизовані процеси резервного копіювання. Основна серверна логіка гри реалізована на базі AWS із використанням сервісу Amazon ECS (Elastic Container Service) та Redis-кешу через Amazon ElastiCache. Для балансування навантаження розглядалися Nginx та AWS Elastic Load Balancer, що дало змогу забезпечити горизонтальне масштабування під час зростання кількості одночасних підключень.

Розгортання серверної інфраструктури здійснено на базі віртуальних Instance сервісу EC2 від AWS із використанням Docker-контейнерів. Під час тестування в умовах, наближених до продакшн-середовища, зафіксовано незначні затримки при роботі з обмеженими ресурсами, однак загальна стабільність та працездатність системи підтверджена. Зокрема, реалізовано ізольовану багаторівневу архітектуру, що дозволяє масштабувати застосунок без потреби у його повній перебудові.

У результаті виконаної роботи доведено ефективність використання хмарних обчислювальних платформ та технологій контейнеризації у процесі розробки масштабованих ігрових вебзастосунків. Побудована архітектура забезпечує гнучкість, адаптивність до змін навантаження та зручність супроводу, що відкриває перспективи для подальшого розвитку та комерціалізації створеної системи відповідно до вимог сучасного програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker Documentation. URL: https://docs.docker.com/ (дата звернення 27.05.2025)
2. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. O'Reilly Media, 2014.
3. Kubernetes Documentation. URL: https://kubernetes.io/docs/ (дата звернення 27.05.2025)
4. Pahl, C. Containerization and the PaaS Cloud, IEEE Cloud Computing, 2015.
5. Hightower, K., Burns, B., Beda, J. Kubernetes: Up and Running. O'Reilly Media, 2017.
6. Merkel, D. Docker: lightweight Linux containers for consistent development and deployment, Linux Journal, 2014.
7. React Documentation. URL: https://reactjs.org/docs/getting-started.html (дата звернення 27.05.2025)
8. Паль, К. Контейнеризація та хмарні платформи PaaS. IEEE Cloud Computing, 2015.
9. Django Documentation. URL: https://docs.djangoproject.com/en/stable/ (дата звернення 27.05.2025)
10. Django Channels Documentation. URL: https://channels.readthedocs.io/en/latest/ (дата звернення 27.05.2025)
11. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення 27.05.2025)
12. Richards, M., Ford, N. Fundamentals of Software Architecture. O'Reilly Media, 2020.
13. Flanagan, D. JavaScript: The Definitive Guide. O'Reilly Media, 2020.
14. RESTful API Design – Best Practices in API Design with REST. URL: https://www.restapitutorial.com/ (дата звернення 27.05.2025)

15. Dockerfile Best Practices. URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (дата звернення 27.05.2025)

16. PixiJS Documentation. URL: https://pixijs.com/ (дата звернення 27.05.2025)

17. Redis Documentation. URL: https://redis.io/documentation (дата звернення 27.05.2025)

18. PostgreSQL Official Documentation. URL: https://www.postgresql.org/docs/ (дата звернення 27.05.2025)

19. Бернштейн, Д. Контейнери і хмара: від LXC до Docker і Kubernetes. O'Reilly Media, 2014.

20. Фланаган, Д. *JavaScript: Вичерпний посібник.* O'Reilly Media, 2020.

21. AWS Elastic Container Service (ECS) Overview. URL: https://aws.amazon.com/ecs/ (дата звернення 27.05.2025)

22. Google Kubernetes Engine (GKE) Overview. URL: https://cloud.google.com/kubernetes-engine (дата звернення 27.05.2025)

23. Azure Kubernetes Service (AKS) Overview. URL: https://learn.microsoft.com/en-us/azure/aks/ (дата звернення 27.05.2025)